

AOSO-LogitBoost: Adaptive One-Vs-One LogitBoost for Multi-Class Problem

Peng Sun

SUNP08@MAILS.TSINGHUA.EDU.CN

Tsinghua National Laboratory for Information Science and Technology(TNList), Department of Automation, Tsinghua University, Beijing 100084, China

Mark D. Reid

MARK.REID@ANU.EDU.AU

Research School of Computer Science, The Australian National University and NICTA, Canberra, Australia

Jie Zhou

JZHOU@TSINGHUA.EDU.CN

Tsinghua National Laboratory for Information Science and Technology(TNList), Department of Automation, Tsinghua University, Beijing 100084, China

Abstract

This paper presents an improvement to model learning when using multi-class LogitBoost for classification. Motivated by the statistical view, LogitBoost can be seen as additive tree regression. Two important factors in this setting are: 1) coupled classifier output due to a sum-to-zero constraint, and 2) the dense Hessian matrices that arise when computing tree node split gain and node value fittings. In general, this setting is too complicated for a tractable model learning algorithm. However, too aggressive simplification of the setting may lead to degraded performance. For example, the original LogitBoost is outperformed by ABC-LogitBoost due to the latter's more careful treatment of the above two factors.

In this paper we propose techniques to address the two main difficulties of the LogitBoost setting: 1) we adopt a vector tree (*i.e.*, each node value is vector) that enforces a sum-to-zero constraint, and 2) we use an adaptive block coordinate descent that exploits the dense Hessian when computing tree split gain and node values. Higher classification accuracy and faster convergence rates are observed for a range of public data sets when compared to both the original and the ABC-LogitBoost implementations.

1. Introduction

Boosting is successful for both binary and multi-class classification (Freund & Schapire, 1995; Schapire & Singer, 1999). Among those popular variants, we are particularly focusing on LogitBoost (Friedman et al., 1998) in this paper. Originally, LogitBoost is motivated by statistical view (Friedman et al., 1998), where boosting algorithms consists of three key components: the loss, the function model, and the optimization algorithm. In the case of LogitBoost, these are the Logit loss, the use of additive tree models, and a stage-wise optimization, respectively. There are two important factors in the LogitBoost setting. Firstly, the posterior class probability estimate must be normalised so as to sum to one in order to use the Logit loss. This leads to a coupled classifier output, *i.e.*, the sum-to-zero classifier output. Secondly, a dense Hessian matrix arises when deriving the tree node split gain and node value fitting. It is challenging to design a tractable optimization algorithm that fully handles both these factors. Consequently, some simplification and/or approximation is needed. Friedman et al. (1998) proposes a “one scalar regression tree for one class” strategy. This breaks the coupling in the classifier output so that at each boosting iteration the model updating collapses to K independent regression tree fittings, where K denotes the number of classes. In this way, the sum-to-zero constraint is dropped and the Hessian is approximated diagonally.

Unfortunately, Friedman's prescription turns out to have some drawbacks. A later improvement, ABC-LogitBoost, is shown to outperform LogitBoost in terms of both classification accuracy and conver-

Appearing in *Proceedings of the 29th International Conference on Machine Learning*, Edinburgh, Scotland, UK, 2012. Copyright 2012 by the author(s)/owner(s).

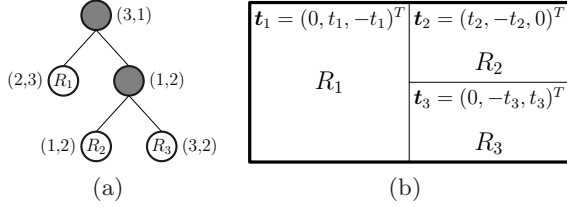


Figure 1. A newly added tree at some boosting iteration for a 3-class problem. (a) A class pair (shown in brackets) is selected for each tree node. For each internal node (filled), the pair is for computing split gain; For terminal nodes (unfilled), it is for node vector updating. (b) The feature space (the outer black box) is partitioned by the tree in (a) into regions $\{R_1, R_2, R_3\}$. On each region only two coordinates are updated based on the corresponding class pair shown in (a).

gence rate (Li, 2008; 2010a). This is due to ABC-LogitBoost’s careful handling of the above key problems of the LogitBoost setting. At each iteration, the sum-to-zero constraint is enforced so that only $K - 1$ scalar trees are fitted for $K - 1$ classes. The remaining class – called the base class – is selected adaptively per iteration (or every several iterations), hence the acronym ABC (Adaptive Base Class). Also, the Hessian matrix is approximated in a more refined manner than the original LogitBoost when computing the tree split gain and fitting node value.

In this paper, we propose two novel techniques to address the challenging aspects of the LogitBoost setting. In our approach, one vector tree is added per iteration. We allow a K dimensional sum-to-zero vector to be fitted for each tree node. This permits us to explicitly formulate the computation for both node split gain and node value fitting as a K dimensional constrained quadratic optimization, which arises as a subproblem in the inner loop for split seeking when fitting a new tree. To avoid the difficulty of a dense Hessian, we propose that for **each** of these subproblems, only two coordinates (*i.e.*, two classes or a class pair) are adaptively selected for updating, hence the name AOSO (Adaptive One vS One) LogitBoost. Figure 1 gives an overview of our approach. In Section 2.5 we show that first order and second order approximation of loss reduction can be a good measure for the quality of selected class pair.

Following the above formulation, ABC-LogitBoost, although derived from a somewhat different framework in (Li, 2010a), can thus be shown to be a special case of AOSO-LogitBoost, but with a less flexible tree model. In Section 3.1 we compare the differences between the two approaches in detail and provide some intuition

for AOSO’s improvement over ABC.

The rest of this paper is organised as follows: In Section 2 we first formulate the problem setting for LogitBoost and then give the details of our approach. In Section 3 we compare our approach to (ABC-)LogitBoost. In Section 4, experimental results in terms of classification errors and convergence rates are reported on a range of public datasets.

2. The Algorithm

We begin with the basic setting for the LogitBoost algorithm. For K -class classification ($K \geq 2$), consider an N example training set $\{\mathbf{x}_i, y_i\}_{i=1}^N$ where \mathbf{x}_i denotes a feature value and $y_i \in \{1, \dots, K\}$ denotes a class label. Class probabilities conditioned on \mathbf{x} , denoted by $\mathbf{p} = (p_1, \dots, p_K)^T$, are learned from the training set. For a test example with known \mathbf{x} and unknown y , we predict a class label by using the Bayes rule: $y = \arg \max_k p_k, k = 1, \dots, K$.

Instead of learning the class probability directly, one learns its “proxy” $\mathbf{F} = (F_1, \dots, F_K)^T$ given by the so-called Logit link function:

$$p_k = \frac{\exp(F_k)}{\sum_{j=1}^K \exp(F_j)} \quad (1)$$

with the constraint $\sum_{k=1}^K F_k = 0$ (Friedman et al., 1998). For simplicity and without confusion, we hereafter omit the dependence on \mathbf{x} for \mathbf{F} and for other related variables.

The \mathbf{F} is obtained by minimizing a target function on training data:

$$Loss = \sum_{i=1}^N L(y_i, \mathbf{F}_i), \quad (2)$$

where \mathbf{F}_i is shorthand for $\mathbf{F}(\mathbf{x}_i)$ and $L(y_i, \mathbf{F}_i)$ is the Logit loss for a single training example:

$$L(y_i, \mathbf{F}_i) = - \sum_{k=1}^K r_{ik} \log p_{ik}, \quad (3)$$

where $r_{ik} = 1$ if $y_i = k$ and 0 otherwise. The probability p_{ik} is connected to F_{ik} via (1).

To make the optimization of (2) feasible, a model is needed to describe how \mathbf{F} depends on \mathbf{x} . For example, linear model $\mathbf{F} = \mathbf{W}^T \mathbf{x}$ is used in traditional Logit regression, while Generalized Additive Model is adopted in LogitBoost:

$$\mathbf{F}(\mathbf{x}) = \sum_{m=1}^M \mathbf{f}_m(\mathbf{x}), \quad (4)$$

where each $\mathbf{f}_m(\mathbf{x})$, a K dimensional sum-to-zero vector, is learned by greedy stage-wise optimization. That is, at each iteration $\mathbf{f}_m(\mathbf{x})$ is added only based on $\mathbf{F} = \sum_{j=1}^{m-1} \mathbf{f}_j$. Formally,

$$\begin{aligned} \mathbf{f}_m(\mathbf{x}) &= \arg \min_{\mathbf{f}} \sum_{i=1}^N L(y_i, \mathbf{F}_i + \mathbf{f}(\mathbf{x}_i)) \\ \text{s.t. } \sum_k f_k(\mathbf{x}_i) &= 0, \quad i = 1, \dots, N. \end{aligned} \quad (5)$$

This procedure repeats M times with initial condition $\mathbf{F} = 0$. Owing to its iterative nature, we only need to know how to solve (5) in order to implement the optimization.

2.1. Vector Tree Model

The $\mathbf{f}(\mathbf{x})$ in (5) is typically represented by K scalar regression trees (*e.g.*, in LogitBoost (Friedman et al., 1998) or the Real AdaBoost.MH implementation in (Friedman et al., 1998)) or a single vector tree (*e.g.*, the Real AdaBoost.MH implementation in (Kégl & Busa-Fekete, 2009)). In this paper, we adopt a single vector tree. We further restrict that it is a binary tree (*i.e.*, only binary splits on internal node are allowed) and the split must be vertical to coordinate axis, as in (Friedman et al., 1998) or (Li, 2010a). Formally,

$$\mathbf{f}(\mathbf{x}) = \sum_{j=1}^J \mathbf{t}_j I(\mathbf{x} \in R_j) \quad (6)$$

where $\{R_j\}_{j=1}^J$ describes how the feature space is partitioned, while $\mathbf{t}_j \in \mathbb{R}^K$ with a sum-to-zero constraint is the node values/vector associated with R_j . See Figure 1 for an example.

2.2. Tree Building

Solving (5) with the tree model (6) is equivalent to determining the parameters $\{\mathbf{t}_j, R_j\}_{j=1}^J$ at the m -th iteration. In this subsection we will show how this problem reduces to solving a collection of convex optimization subproblems for which we can use any numerical method. Following Friedman's LogitBoost settings, here we use Newton descent¹. Also, we will show how the gradient and Hessian can be computed incrementally.

We begin with some shorthand notation for the *node*

¹We use Newtown descent as there is evidence in (Li, 2010a) that gradient descent, *i.e.*, in Friedmans's MART (Friedman, 2001), leads to decreased classification accuracy.

loss:

$$\begin{aligned} \text{NodeLoss}(\mathbf{t}; \mathcal{I}) &= \sum_{i \in \mathcal{I}} L(y_i, \mathbf{F}_i + \mathbf{t}) \\ t_1 + \dots + t_K &= 0, \quad \mathbf{t} \in \mathbb{R}^K \end{aligned} \quad (7)$$

where \mathcal{I} denotes the index set of the training examples on some either internal or terminal node (*i.e.*, those falling into the corresponding region). Minimizing (7) is the bridge to $\{\mathbf{t}_j, R_j\}$ in that:

1. To obtain $\{\mathbf{t}_j\}$ with given $\{R_j\}$, we simply take the minimizer of (7):

$$\mathbf{t}_j = \arg \min_{\mathbf{t}} \text{NodeLoss}(\mathbf{t}; \mathcal{I}_j), \quad (8)$$

where \mathcal{I}_j denotes the index set for R_j .

2. To obtain $\{R_j\}$, we recursively perform binary split until there are J -terminal nodes.

The key to the second point is to explicitly give the node split gain. Suppose an internal node with n training examples ($n = N$ for the root node), we fix on some feature and re-index all the n examples according to their sorted feature values. Now we need to find the index n' with $1 < n' < n$ that maximizes the node gain defined as loss reduction after a division between the n' -th and $(n' + 1)$ -th examples:

$$\begin{aligned} \text{NodeGain}(n') &= \text{NodeLoss}(\mathbf{t}^*; \mathcal{I}) - \\ &(\text{NodeLoss}(\mathbf{t}_L^*; \mathcal{I}_L) + \text{NodeLoss}(\mathbf{t}_R^*; \mathcal{I}_R)) \end{aligned} \quad (9)$$

where $\mathcal{I} = \{1, \dots, n\}$, $\mathcal{I}_L = \{1, \dots, n'\}$ and $\mathcal{I}_R = \{n' + 1, \dots, n\}$; \mathbf{t}^* , \mathbf{t}_L^* and \mathbf{t}_R^* are the minimizers of (7) with index sets \mathcal{I} , \mathcal{I}_L and \mathcal{I}_R , respectively. Generally, this searching applies to all features. The best division resulting to largest (9) is then recorded to perform the actual node split.

Note that (9) arises in the context of an $O(N \times D)$ outer loop, where D is number of features. However, a naïve summing of the losses for (7) incurs an additional $O(N)$ factor in complexity, which finally results in an unacceptable $O(N^2 D)$ complexity for a single boosting iteration.

A workaround is to use a Newton descent method for which both the gradient and Hessian can be incrementally computed. Let \mathbf{g} , \mathbf{H} respectively be the $K \times 1$ gradient vector and $K \times K$ Hessian matrix at $\mathbf{t} = \mathbf{0}$. By dropping the constant $\text{NodeLoss}(\mathbf{0}; \mathcal{I})$ that is irrelevant to \mathbf{t} , the Taylor expansion of (7) w.r.t. \mathbf{t} up to second order is:

$$\begin{aligned} \text{loss}(\mathbf{t}; \mathcal{I}) &= \mathbf{g}^T \mathbf{t} + \frac{1}{2} \mathbf{t}^T \mathbf{H} \mathbf{t} \\ t_1 + \dots + t_K &= 0, \quad \mathbf{t} \in \mathbb{R}^K \end{aligned} \quad (10)$$

By noting the additive separability of (10) and using some matrix derivatives, we have

$$\mathbf{g} = -\sum_{i \in I} \mathbf{g}_i \quad (11) \quad \mathbf{H} = \sum_{i \in I} \mathbf{H}_i \quad (12)$$

$$\mathbf{g}_i = \mathbf{r}_i - \mathbf{p}_i \quad (13) \quad \mathbf{H}_i = \hat{\mathbf{P}} - \mathbf{p}_i \mathbf{p}_i^T \quad (14)$$

where the diagonal matrix $\hat{\mathbf{P}} = \text{diag}(p_{i1}, \dots, p_{iK})$. We then use (10) to compute the approximated node loss in (9). Thanks to the additive form, both (11) and (12) can be incrementally/decrementally computed in constant time when the split searching proceeds from one training example to the next. Therefore, the computation of (10) eliminates the $O(N)$ complexity in the naïve summing of losses.²

2.3. Properties of Approximated Node Loss

To minimise (10), we give some properties for (10) that should be taken into account when seeking a solver. We begin with the sum-to-zero constraint. The probability estimate p_k in the Logit loss (3) must be non-zero and sum-to-one, which is ensured by the link function (1). Such a link, in turn, means that p_k is unchanged by adding an arbitrary constant to each component in \mathbf{F} . As a result, the single example loss (3) is invariant to moving it along an all-1 vector $\mathbf{1}$. That is,

$$L(y_i, \mathbf{F}_i + c\mathbf{1}) = L(y_i, \mathbf{F}_i), \quad (15)$$

where c is an arbitrary real constant (Note that $\mathbf{1}$ is, coincidentally, the orthogonal complement to the space defined by sum-to-zero constraint). This property also carries over to the approximated node loss (10):

Property 1 $\text{loss}(\mathbf{t}; \mathcal{I}) = \text{loss}(\mathbf{t} + c\mathbf{1}; \mathcal{I})$.

This is obvious by noting the additive separability in (10), as well as that $\mathbf{g}_i^T \mathbf{1} = 0$, $\mathbf{1}^T \mathbf{H}_i \mathbf{1} = 0$ holds since \mathbf{p}_i is sum-to-one.

For the Hessian, we have $\text{rank}(\mathbf{H}) \leq \text{rank}(\mathbf{H}_i)$ by noting the additive form in (11). In (Li, 2010a) it is shown that $\det \mathbf{H}_i = 0$ by brute-force determinant expansion. Here we give a stronger property:

²In Real AdaBoost.MH, such a second order approximation is not necessary (although possible, cf. (Zou et al., 2008)). Due to the special form of the exponential loss and the absence of a sum-to-zero constraint, there exists analytical solution for the node loss (7) by simply setting the derivative to 0. Here also, the computation can be incremental/decremental. Since the loss design and AdaBoost.MH are not our main interests, we do not discuss this further.

Property 2 \mathbf{H}_i is a positive semi-definite matrix such that 1) $\text{rank}(\mathbf{H}_i) = \kappa - 1$, where κ is the number of non-zero elements in \mathbf{p}_i ; 2) $\mathbf{1}$ is the eigenvector for eigenvalue 0.

The proof can be found in this paper's extended version (Sun et al., 2012).

The properties shown above indicate that 1) \mathbf{H} is singular, so that unconstrained Newton descent is not applicable here, and 2) $\text{rank}(\mathbf{H})$ could be as high as $K - 1$, which prohibits the application of the standard fast quadratic solver designed for low rank Hessians. In the following we propose to address this problem via block coordinate descent, a technique that has been successfully used in training SVMs (Bottou & Lin, 2007).

2.4. Block Coordinate Descent

For the variable \mathbf{t} in (10), we only choose two (the least possible number due to the sum-to-zero constraint) coordinates, *i.e.*, a class pair, to update while keeping the others fixed. Suppose we have chosen the r -th and the s -th coordinate (how to do so is deferred to next subsection). Let $t_r = t$ and $t_s = -t$ be the free variables (such that $t_r + t_s = 0$) and $t_k = 0$ for $k \neq r, k \neq s$. Plugging these into (10) yields an unconstrained one dimensional quadratic problem with regards to the scalar variable t :

$$\text{loss}(t) = g^T t + \frac{1}{2} h t^2 \quad (16)$$

where the gradient and Hessian collapse to scalars:

$$g = -\sum_{i \in I} ((r_{i,r} - p_{i,r}) - (r_{i,s} - p_{i,s})) \quad (17)$$

$$h = \sum_{i \in I} (p_{i,r}(1 - p_{i,r}) + p_{i,s}(1 - p_{i,s}) + 2p_{i,r}p_{i,s}), \quad (18)$$

To this extent, we are able to obtain the analytical expression for the minimizer and minimum of (16):

$$t^* = \arg \min_t \text{loss}(t) = -\frac{g}{h} \quad (19)$$

$$\text{loss}(t^*) = -\frac{g^2}{2h} \quad (20)$$

by noting the non-negativity of (18).

Based on (19), node vector (8) can be approximated as

$$\mathbf{t}_{mj} = \begin{cases} +(-g/h) & k = r \\ -(-g/h) & k = s \\ 0 & \text{otherwise} \end{cases} \quad (21)$$

where g and h are respectively computed by using (17) and (18) with index set \mathcal{I}_{mj} . Based on (20), the node gain (9) can be approximated as

$$\text{NodeGain}(n') = \frac{g_L^2}{2h_L} + \frac{g_R^2}{2h_R} - \frac{g^2}{2h}, \quad (22)$$

where g (or g_L, g_R) and h (or h_L, h_R) are computed by using (17) and (18) with index set \mathcal{I} (or $\mathcal{I}_L, \mathcal{I}_R$).

2.5. Class Pair Selection

In (Bottou & Lin, 2007) two methods for selecting (r, s) are proposed. One is based on a first order approximation. Let t_r and t_s be the free variables and the rest be fixed to 0. For a \mathbf{t} with sufficiently small fixed length, let $t_r = \epsilon$ and $t_s = -\epsilon$ where $\epsilon > 0$ is some small enough constant. The first order approximation of (10) is:

$$\text{loss}(\mathbf{t}) \approx \text{loss}(\mathbf{0}) + \mathbf{g}^T \mathbf{t} = \text{loss}(\mathbf{0}) - \epsilon(-g_r - (-g_s)) \quad (23)$$

It follows that the indices r, s resulting in largest decrement to (23) are:

$$\begin{aligned} r &= \arg \max_k \{-g_k\} \\ s &= \arg \min_k \{-g_k\}. \end{aligned} \quad (24)$$

Another method that can be derived in a similar way takes into account the second order information:

$$\begin{aligned} r &= \arg \max_k \{-g_k\} \\ s &= \arg \max_k \left\{ \frac{(g_r - g_k)^2}{h_{rr} + h_{kk} - 2h_{rk}} \right\}, \end{aligned} \quad (25)$$

Both methods are $O(K)$ procedures that are better than the $K \times (K - 1)/2$ naïve enumeration. However, in our implementation we find that (25) achieves better results for AOSO-LogitBoost.

Pseudocode for AOSO-LogitBoost is given in Algorithm 1.

3. Comparison to (ABC-)LogitBoost

In this section we compare the derivations of LogitBoost and ABC-LogitBoost and provide some intuition for observed behaviours in the experiments in Section 4.

3.1. ABC-LogitBoost

To solve (5) with a sum-to-zero constraint, ABC-LogitBoost uses $K - 1$ independent trees:

$$f_k = \begin{cases} \sum_j t_{jk} I(x \in R_{jk}) & k \neq b \\ -\sum_{l \neq b} f_l & k = b. \end{cases} \quad (26)$$

Algorithm 1 AOSO-LogitBoost. v is shrinkage factor that controls learning rate.

```

1:  $F_{ik} = 0, \quad k = 1, \dots, K, i = 1, \dots, N$ 
2: for  $m = 1$  to  $M$  do
3:    $p_{i,k} = \frac{\exp(F_{i,k})}{\sum_{j=1}^K \exp(F_{i,j})}, k = 1, \dots, K, i = 1, \dots, N.$ 
4:   Obtain  $\{R_{mj}\}_{j=1}^J$  by recursive region partition.
      Node split gain is computed as (22), where the
      class pair  $(r, s)$  is selected using (25).
5:   Compute  $\{t_{mj}\}_{j=1}^J$  by (21), where the class pair
       $(r, s)$  is selected using (25).
6:    $\mathbf{F}_i = \mathbf{F}_i + v \sum_{j=1}^J t_{mj} I(\mathbf{x}_i \in R_{mj}), \quad i =$ 
       $1, \dots, N.$ 
7: end for
    
```

In (Li, 2010a), the so-called base class b is selected by exhaustive search per iteration, *i.e.*, trying all possible b , which involves growing $K(K - 1)$ trees. To reduce the time complexity, Li also proposed other methods. In (Li, 2010c), b is selected only every several iterations, while in (Li, 2008), b is, intuitively, set to be the class that leads to largest loss reduction at last iteration.

In ABC-LogitBoost the sum-to-zero constraint is explicitly considered when deriving the node value and the node split gain for the scalar regression tree. Indeed, they are the same as (21) and (22) in this paper, although derived using a slightly different motivation. In this sense, ABC-LogitBoost can be seen as a special form of the AOSO-LogitBoost since: 1) For each tree, the class pair is fixed for every node in ABC, while it is selected adaptively in AOSO, and 2) $K - 1$ trees are added per iteration in ABC (using the same set of probability estimates $\{p_i\}_{i=1}^N$), while only one tree is added per iteration by AOSO (and $\{p_i\}_{i=1}^N$ are updated as soon as each tree is added).

Since two changes are made to ABC-LogitBoost, an immediate question is what happens if we only make one? That is, what happens if one vector tree is added per iteration for a single class pair selected only for the root node and shared by all other tree nodes, as in ABC, but the $\{p_i\}_{i=1}^N$ are updated as soon as a tree is added, as in AOSO. This was tried but unfortunately, **degraded performance** was observed for this combination so the results are not reported here.

From the above analysis, we believe the more flexible model (as well as the model updating strategy) in AOSO is what contributes to its improvement over ABC, as seen section 4).

Table 1. Datasets used in our experiments.

datasets	K	#features	#training	#test
Poker525k	10	25	525010	500000
Poker275k	10	25	275010	500000
Poker150k	10	25	150010	500000
Poker100k	10	25	100010	500000
Poker25kT1	10	25	25010	500000
Poker25kT2	10	25	25010	500000
Coverttype290k	7	54	290506	290506
Coverttype145k	7	54	145253	290506
Letter	26	16	16000	4000
Letter15k	26	16	15000	5000
Letter2k	26	16	2000	18000
Letter4K	26	16	4000	16000
Pendigits	10	16	7494	3498
Zipcode	10	256	7291	2007
(a.k.a. USPS)				
Isolet	26	617	6238	1559
Optdigits	10	64	3823	1797
Mnist10k	10	784	10000	60000
M-Basic	10	784	12000	50000
M-Image	10	784	12000	50000
M-Rand	10	784	12000	50000
M-Noise1	10	784	10000	2000
M-Noise2	10	784	10000	2000
M-Noise3	10	784	10000	2000
M-Noise4	10	784	10000	2000
M-Noise5	10	784	10000	2000
M-Noise6	10	784	10000	2000

3.2. LogitBoost

In the original LogitBoost (Friedman et al., 1998), the Hessian matrix (14) is approximated diagonally. In this way, the \mathbf{f} in (5) is expressed by K uncoupled scalar tress:

$$f_k = \sum_j t_{jk} I(x \in R_{jk}), \quad k = 1, 2, \dots, K \quad (27)$$

with the gradient and Hessian for computing node value and node split gain given by:

$$g_k = - \sum_{i \in \mathcal{I}} (r_{i,k} - p_{i,k}), \quad h_k = - \sum_{i \in \mathcal{I}} p_{i,k} (1 - p_{i,k}). \quad (28)$$

Here we use the subscript k for g and h to emphasize the k -th tree is built independently to the other $K - 1$ trees (*i.e.*, the sum-to-zero constraint is dropped). Although this simplifies the mathematics, such an aggressive approximation turns out to harm both classification accuracy and convergence rate, as shown in Li’s experiments (Li, 2009).

4. Experiments

In this section we compare AOSO-LogitBoost with ABC-LogitBoost, which was shown to outperform original LogitBoost in Li’s experiments (Li, 2010a; 2009). We test AOSO on all the datasets used in (Li, 2010a; 2009), as listed in Table 1. In the top section are UCI datasets and in the bottom are Mnist datasets with many variations (see (Li, 2010b)

Table 2. Test classification errors on *Mnist10k*. In each J - v entry, the first entry is for ABC-LogitBoost and the second for AOSO-LogitBoost. Lower one is in bold.

	$v = 0.04$	$v = 0.06$	$v = 0.08$	$v = 0.1$
$J = 4$	2630 2515	2600 2414	2535 2414	2522 2392
$J = 6$	2263 2133	2252 2146	2226 2146	2223 2134
$J = 8$	2159 2055	2138 2046	2120 2046	2143 2055
$J = 10$	2122 2010	2118 1980	2091 1980	2097 2014
$J = 12$	2084 1968	2090 1965	2090 1965	2095 1995
$J = 14$	2083 1945	2094 1938	2063 1938	2050 1935
$J = 16$	2111 1941	2114 1928	2097 1928	2082 1966
$J = 18$	2088 1925	2087 1916	2088 1916	2097 1920
$J = 20$	2128 1930	2112 1917	2095 1917	2102 1948
$J = 24$	2174 1901	2147 1920	2129 1920	2138 1903
$J = 30$	2235 1887	2237 1885	2221 1885	2177 1917
$J = 40$	2310 1923	2284 1890	2257 1890	2260 1912
$J = 50$	2353 1958	2359 1910	2332 1910	2341 1934

for detailed descriptions).³ To exhaust the learning ability of (ABC-)LogitBoost, Li let the boosting stop when either the training converges (*i.e.*, the loss (2) approaches 0, implemented as $\leq 10^{-16}$) or a maximum number of iterations, M , is reached. Test errors at last iteration are simply reported since no obvious over-fitting is observed. By default, $M = 10000$, while for those large datasets (*Cover-type290k*, *Poker525k*, *Poker275k*, *Poker150k*, *Poker100k*) $M = 5000$ (Li, 2010a; 2009). We adopt the same criteria, except that our maximum iterations $M_{AOSO} = (K - 1) \times M_{ABC}$, where K is the number of classes. Note that only one tree is added at each iteration in AOSO, while $K - 1$ are added in ABC. Thus, this correction compares the same maximum number of trees for both AOSO and ABC.

The most important tuning parameters in LogitBoost are the number of terminal nodes J , and the shrinkage factor v . In (Li, 2010a; 2009), Li reported results of (ABC-)LogitBoost for a number of J - v combinations. We report the corresponding results for AOSO-LogitBoost for the same combinations. In the following, we intend to show that **for nearly all J - v combinations, AOSO-LogitBoost has lower classification error and faster convergence rates than ABC-LogitBoost.**

4.1. Classification Errors

Table 2 shows results of various J - v combinations for a representative datasets. Results on more datasets can be found in this paper’s extended version (Sun et al., 2012).

In Table 3 we summarize the results for all datasets. In (Li, 2010a), Li reported that ABC-LogitBoost is insensitive to J , v on all datasets except for *Poker25kT1*

³Code and data are available at <http://ivg.au.tsinghua.edu.cn/index.php?n=People.PengSun>

Table 3. Summary of test classification errors. Lower one is in bold. Middle panel: $J = 20, v = 0.1$ except for *Poker25kT1* and *Poker25kT2* on which J, v are chosen by validation (See the text in 4.1); Right panel: the overall best. Dash “-” means unavailable in (Li, 2010a)(Li, 2009). Relative improvements (R) and P -values (pv) are given.

Datasets	#tests	ABC	AOSO	R	pv	ABC*	AOSO*	R	pv
Poker525k	500000	1736	1537	0.1146	0.0002	-	-	-	-
Poker275k	500000	2727	2624	0.0378	0.0790	-	-	-	-
Poker150k	500000	5104	3951	0.2259	0.0000	-	-	-	-
Poker100k	500000	13707	7558	0.4486	0.0000	-	-	-	-
Poker25kT1	500000	37345	31399	0.1592	0.0000	37345	31399	0.1592	0.0000
Poker25kT2	500000	36731	31645	0.1385	0.0000	36731	31645	0.1385	0.0000
Coverttype290k	290506	9727	9586	0.0145	0.1511	-	-	-	-
Coverttype145k	290506	13986	13712	0.0196	0.0458	-	-	-	-
Letter	4000	89	92	-0.0337	0.5892	89	88	0.0112	0.4697
Letter15k	5000	109	116	-0.0642	0.6815	-	-	-	-
Letter4k	16000	1055	991	0.0607	0.0718	1034	961	0.0706	0.0457
Letter2k	18000	2034	1862	0.0846	0.0018	1991	1851	0.0703	0.0084
Pendigits	3498	100	83	0.1700	0.1014	90	81	0.1000	0.2430
Zipcode	2007	96	99	-0.0313	0.5872	92	94	-0.0217	0.5597
Isolet	1559	65	55	0.1538	0.1759	55	50	0.0909	0.3039
Optdigits	1797	55	38	0.3091	0.0370	38	34	0.1053	0.3170
Mnist10k	60000	2102	1948	0.0733	0.0069	2050	1885	0.0805	0.0037
M-Basic	50000	1602	1434	0.1049	0.0010	-	-	-	-
M-Rotate	50000	5959	5729	0.0386	0.0118	-	-	-	-
M-Image	50000	4268	4167	0.0237	0.1252	4214	4002	0.0503	0.0073
M-Rand	50000	4725	4588	0.0290	0.0680	-	-	-	-
M-Noise1	2000	234	228	0.0256	0.3833	-	-	-	-
M-Noise2	2000	237	233	0.0169	0.4221	-	-	-	-
M-Noise3	2000	238	233	0.0210	0.4031	-	-	-	-
M-Noise4	2000	238	233	0.0210	0.4031	-	-	-	-
M-Noise5	2000	227	214	0.0573	0.2558	-	-	-	-
M-Noise6	2000	201	191	0.0498	0.2974	-	-	-	-

Table 4. #trees added when convergence on selected datasets. R stands for the ratio AOSO/ABC.

	Mnist10k	M-Rand	M-Image
ABC	7092	15255	14958
R	0.7689	0.7763	0.8101
	Letter15k	Letter4k	Letter2k
ABC	45000	20900	13275
R	0.5512	0.5587	0.5424

and *Poker25kT2*. Therefore, Li summarized classification errors for ABC simply with $J = 20$ and $v = 0.1$, except that on *Poker25kT1* and *Poker25kT2* errors are reported by using the other’s test set as a validation set. Based on the same criteria we summarize AOSO in the middle panel of Table 3 where the test errors as well as the improvement relative to ABC are given. In the right panel of Table 3 we provide the comparison for the best results achieved over all $J-v$ combinations when the corresponding results for ABC are available in (Li, 2010a) or (Li, 2009).

We also tested the statistical significance between AOSO and ABC. We assume the classification error rate is subject to some Binomial distribution. Let z denote the number of errors and n the number of tests, then the estimate of error rate $\hat{p} = z/n$ and its variance is $\hat{p}(1 - \hat{p})/n$. Subsequently, we approximate the Binomial distribution by a Gaussian distribution and perform a hypothesis test. The p -values are reported in Table 3.

For some problems, we note LogitBoost (both ABC

Table 5. #trees added when convergence on *Mnist10k* for a number of $J-v$ combinations. For each $J-v$ entry, the first number is for ABC, the second for the ratio AOSO/ABC.

	$v = 0.04$	$v = 0.06$	$v = 0.1$
$J = 4$	90000 1.0	90000 1.0	90000 1.0
$J = 6$	90000 0.7740	63531 0.7249	38223 0.7175
$J = 8$	55989 0.7962	38223 0.7788	22482 0.7915
$J = 10$	39780 0.8103	27135 0.7973	16227 0.8000
$J = 12$	31653 0.8109	20997 0.8074	12501 0.8269
$J = 14$	26694 0.7854	17397 0.8047	10449 0.8160
$J = 16$	22671 0.7832	11704 0.8290	8910 0.8063
$J = 18$	19602 0.7805	13104 0.7888	7803 0.7933
$J = 20$	17910 0.7706	11970 0.7683	7092 0.7689
$J = 24$	14895 0.7514	9999 0.7567	6012 0.7596
$J = 30$	12168 0.7333	8028 0.7272	4761 0.7524
$J = 40$	9846 0.6750	6498 0.6853	3870 0.6917
$J = 50$	8505 0.6420	5571 0.6448	3348 0.6589

and AOSO) outperforms other state-of-the-art classifier such as SVM or Deep Learning. (e.g., the test error rate on *Poker* is 40% for SVM and < 10% for both ABC and AOSO (even lower than ABC); on *M-Image* it is 16.15% for DBN-1, 8.54% for ABC and 8.33% for AOSO). See this paper’s extended version (Sun et al., 2012) for details. This shows that the AOSO’s improvement over ABC does deserve the efforts.

4.2. Convergence Rate

Recall that we stop the boosting procedure if either the maximum number of iterations is reached or it converges (i.e. the loss (2) $\leq 10^{-16}$). The fewer trees added when boosting stops, the faster the convergence and the lower the time cost for either training or test-

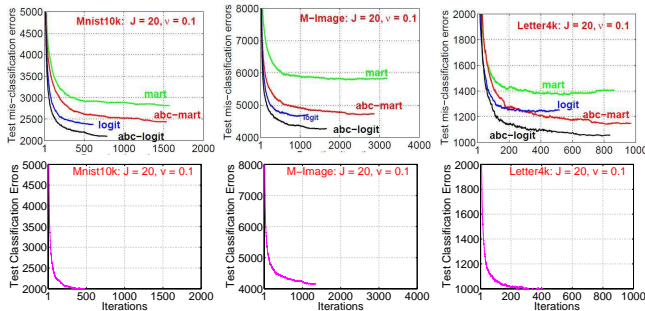


Figure 2. Errors vs. iterations on selected datasets and parameters. Top row: ABC (copied from (Li, 2010a)); Bottom row: AOSO (horizontal axis scaled to compensate the $K - 1$ factor).

ing. We compare AOSO with ABC in terms of the number of trees added when boosting stops for the results of ABC available in (Li, 2010a; 2009). Note that simply comparing number of boosting iterations is unfair to AOSO, since at each iteration only one tree is added in AOSO and $K - 1$ in ABC.

Results are shown in Table 4 and Table 5. Except for when $J-v$ is too small, or particularly difficult datasets where both ABC and AOSO reach maximum iterations, we found that trees needed in AOSO are typically only 50% to 80% of those in ABC.

Figure 2 shows plots for test classification error vs. iterations in both ABC and AOSO and show that AOSO’s test error decreases faster. More plots for AOSO can be found in this paper’s extended version (Sun et al., 2012).

5. Conclusions

We present an improved LogitBoost, namely AOSO-LogitBoost, for multi-class classification. Compared with ABC-LogitBoost, our experiments suggest that our adaptive class pair selection technique results in lower classification error and faster convergence rates.

ACKNOWLEDGMENTS

We appreciate Ping Li’s inspiring discussion and generous encouragement. Comments from NIPS2011 and ICML2012 anonymous reviewers helped improve the readability of this paper. This work was supported by National Natural Science Foundation of China (61020106004, 61021063, 61005023), The National Key Technology R&D Program (2009BAH40B03). NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the ARC

through the ICT Centre of Excellence program.

References

- Bottou, L. and Lin, C.-J. Support vector machine solvers. In Bottou, L., Chapelle, O., DeCoste, D., and Weston, J. (eds.), *Large Scale Kernel Machines*, pp. 301–320. MIT Press, Cambridge, MA., 2007. URL <http://leon.bottou.org/papers/bottou-lin-2006>.
- Freund, Y. and Schapire, R. A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pp. 23–37. Springer, 1995.
- Friedman, J. Greedy function approximation: a gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.
- Friedman, J., Hastie, T., and Tibshirani, R. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28(2):337–407, 1998.
- Kégl, B. and Busa-Fekete, R. Boosting products of base classifiers. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 497–504. ACM, 2009.
- Li, P. Adaptive base class boost for multi-class classification. *Arxiv preprint arXiv:0811.1250*, 2008.
- Li, P. Abc-logitboost for multi-class classification. *Arxiv preprint arXiv:0908.4144*, 2009.
- Li, P. Robust logitboost and adaptive base class (abc) logitboost. In *Conference on Uncertainty in Artificial Intelligence*, 2010a.
- Li, P. An empirical evaluation of four algorithms for multi-class classification: Mart, abc-mart, robust logitboost, and abc-logitboost. *Arxiv preprint arXiv:1001.1020*, 2010b.
- Li, P. Fast abc-boost for multi-class classification. *Arxiv preprint arXiv:1006.5051*, 2010c.
- Schapire, R. and Singer, Y. Improved boosting algorithms using confidence-rated predictions. *Machine learning*, 37(3):297–336, 1999.
- Sun, P., Reid, M., and Zhou, J. Aoso-logitboost: Adaptive one-vs-one logitboost for multi-class problem. *Arxiv preprint arXiv:1110.3907*(<http://arxiv.org/abs/1110.3907>), 2012.
- Zou, H., Zhu, J., and Hastie, T. New multicategory boosting algorithms based on multicategory fisher-consistent losses. *The Annals of Applied Statistics*, 2(4):1290–1306, 2008.